如何实现自己的jQuery

wizardforcel

Published
with GitBook

目錄

介绍 0
讲解基础框架格式 1
初步体验 2
新增next,prev,parent,parents 3
完善init方法 4
新增必备方法each 5
新增find,last,eq,get,first,ajax 6
完善hasClass和css 方法 新增data和attr方法 7
新增html，remove，after，append，before方法 8
引入delegate机制 9
如何实现on与off 10
实现事件委托 11
最后一节补充width,height,extend 12

可想造一个属于你自己的jQuery库？

> 作者：MeCKodo
>
> 来源：forchange

- [x] 0.讲解基础框架格式
- [x] 1.初步体验
- [x] 2.新增next,prev,parent,parents
- [x] 3.完善init方法
- [x] 4.新增必备方法each
- [x] 5.新增find,last,eq,get,first,ajax
- [x] 6.完善hasClass和css 方法 新增data和attr方法
- [x] 7.新增html，remove，after，append，before方法
- [x] 8.引入delegate机制
- [x] 9.如何实现on与off
- [x] 10.实现事件委托
- [x] 11.最后一节补充width,height,extend

前言

> - 1.给一些很想自己实现一个jQuery或者是对实现jQuery非常好奇的人
> - 2.想提升自己js基础的小伙伴
> - 3.本教程系列不考虑兼容和性能问题,只考虑如何利用各种巧妙的方法去实现一个一模一样的API
> - 4.从dom操作一直到事件机制 `on(),off()` ,全会逐一实现(事件机制是本人思考后的另类设计,纯原创)

本人一直很想自己造个jQuery的小库,第一是满足下自己,第二是去体验下jQuery内部的基情!

虽然jQuery很多源码看不懂,但是凭借着对jQuery的API实现的效果,我也基本实现了这样一个类库.

由于自己看别人源码的时候经常会想,作者要是能一步一步的告诉我他是怎么写怎么想的就好了 :).

接下来,我会在每一个version里写下我每一步的想法,让你了解到你如何也能自己造一个这样的轮子.

希望我的做法能给你带来许多的启发.(即使我在里面写的代码实在是不值得一提)

另外您的star,是我的最大动力!

TO DO LIST

- [x] 1.css操作
- [x] 2.class 操作
- [x] 3.attr和data 操作
- [x] 4.简单的dom选择
- [x] 5.dom操作
- [x] 6.ajax
- [x] 7.each循环
- [x] 8.after && before 插入
- [x] 9.事件委托
- [x] 10.tap实现(番外篇)
- [x] 11.简单的一些手势（番外篇）

Lesson-0

===

首先第一个版本,我们要先了解搭建一个库或者是一个给别人使用的小插件应该用一种什么样的格式.

首先我们需要创建一个闭包

```
(function(){
    //code..
})();
```

然后将我们所需要的代码和逻辑都写在里面避免全局变量的泛滥.

接着我们来看看我们第一版里的代码.

```
(function(window,document) {
    var w = window,
        doc = document;
    var Kodo = function(selector) {
        return new Kodo.prototype.init(selector);
    }
    Kodo.prototype = {
        constructor : Kodo,
        length : 0,
        splice: [].splice,
        selector : '',
        init : function(selector) {//dom选择的一些判断

        }
    }
    Kodo.prototype.init.prototype = Kodo.prototype;

    Kodo.ajax = function() { //直接挂载方法  可k.ajax调用
        console.log(this);
    }

    window.f = Kodo;
})(window,document);
```

我创建了一个闭包,传入了window,document并且在内部将他们缓存起来.

接着

```
var kodo = function(selector) {
    return new Kodo.prototype.init(selector);
}
```

如果有看过jQuery源码的童鞋对这个真是在了解不过了.每次用kodo调用的时候,将直接 返回一个kodo的实例.达到无new调用的效果

```
Kodo.prototype = {
    constructor : Kodo,
    length : 0,
    splice: [].splice,
    selector : '',
    init : function(selector) {//dom选择的一些判断

    }
}
Kodo.prototype.init.prototype = Kodo.prototype;
```

接着重点就在于如何去构造Kodo的prototype的原型了.在这上面的属性也就相当于是jQuery的实例方法和属性.所以每次$()后都能链式调用.

由于我们是return new Kodo.prototype.init,那自然,我们需要手动的把init的prototype指向Kodo的prototype

同时我们在原型上具有splice属性后,我们的对象就会变为了一个类数组对象,神奇吧!

```
Kodo.ajax = function() { //直接挂载方法  可f.ajax调用
    console.log(this);
}
```

由于javascript中一切皆对象,所以我们能在我们的Kodo上直接用.XXX来赋予新的属性和方法,这样的方法也被称之为静态方法.

```
window.f = Kodo;
```

最后我们在window上对外暴露一个接口,我们就可以愉快的用 f.ajax 或者是 f("#id") 即可调用.

虽然我很low,但是你看都看完了难道还不给star?!

Lesson-1

===

这个版本呢,先来加四个很简单的方法感受感受下!

首先3个class不用说了

```
hasClass : function(cls) {
    var reg = new RegExp('(\\s|^)' + cls + '(\\s|$)');
    for (var i = 0; i < this.length; i++) {
        if (this[i].className.match(reg)) return true;
            return false;
    }
    return this;
},
addClass : function(cls) {
    var reg = new RegExp('(\\s|^)' + cls + '(\\s|$)');
    for (var i = 0; i < this.length; i++) {
        if(!this[i].className.match(reg))
            this[i].className += ' ' + cls;
    }
    return this;
},
removeClass : function(cls) {
    var reg = new RegExp('(\\s|^)' + cls + '(\\s|$)');
    for (var i = 0; i < this.length; i++) {
        if (this[i].className.match(reg))
            this[i].className = this[i].className.replace(cls,'');
    }
    return this;
}
```

然后新增一个

```
css : function(attr,val) {//链式测试
    console.log(this.length);
    for(var i = 0;i < this.length; i++) {
        if(arguments.length == 1) {
            return getComputedStyle(this[i],null)[attr];
        }
        this[i].style[attr] = val;
    }
    return this;
}
```

这些其实都很简单,我们都要记住,我们封装的DOM对象是一个数组,所以一定都需要用循环来进行各种个样的处理.

然后css这我是用arguments的个数来进行判断是取值还是社值.

最后千万别忘了每个方法的最后都要return this以便链式调用.

大家可以自行拿这几个方法 log出来看看是否是与jQuery的一样就知道是否成功了.

Lesson-2

===

这个版本新增 next(),prev(),parent(),parents()

这4个选择元素的方法还是比较常用的

首先我们需要一个func来过滤我们需要的dom

```
function sibling(cur, dir) {
    while ((cur = cur[dir]) && cur.nodeType !== 1) {}
    return cur;
}
```

上面那段比较简单,就是普通的过滤下元素

```
next : function() {
    return sibling(this[0], "nextSibling");
},
prev : function() {
    return sibling(this[0], "previousSibling");
},
```

看下next方法的源码就知道,我传入Kodo数组对象的0个dom对象,然后取它的下一个同辈元素,直接返回,prev方法同理

```
parent : function() {
    var parent = this[0].parentNode;
    parent && parent.nodeType !== 11 ? parent : null;
    var a = Kodo();
        a[0] = parent;
        a.selector = parent.tagName.toLocaleLowerCase();
        a.length = 1;
    return a;
},
```

这段是取到第一个父元素,由于parent()返回的不是原生的DOM对象,是封装过的数组对象(Kodo),那我们就想办法构造一个新的Kodo对象即可

所以我在里面var了一个 Kodo,然后设置这个Kodo数组对象的selector等配置,然后直接返回这个新的Kodo对象

```
parents : function() {
    var a = Kodo(),
        i = 0;
    while ( (this[0] = this[0][ 'parentNode' ]) && this[0].nodeType
      if ( this[0].nodeType === 1 ) {
        a[i] = this[0];
        i++;
      }
    }
    a.length = i;
    return a;
}
```

同理,在jQuery的parents方法中,返回的依旧是jQuery对象.我们依旧用上面的办法,构造一个新对象并且返回就好了!

中间一层while循环,依次过滤出我们需要的dom元素,然后把他们都赋值到我们新var的对象里,最后别忘了设置一下新对象的length属性,返回我们的新对象即可!

看了上面几个方法是不是觉得!其实很多时候我们完全可以自己新创建一个对象,然后配置好它直接返回这个新对象.比如find方法我们也可以用这样的办法:)

Lesson-3

===

修改f(selector) 里的判断,新增domReady

我们知道在jQuery中还有一种选择器写法

```
$(function() {

});
```

在dom加载完毕后马上就执行,这样的方法会比onload更快,所以domReady对于我们来说一定是必不可少的

我们在init方法中要新增以下判断

```
if(!selector) { return this; }

if (typeof selector == 'object') {
    var selector = [selector];
    for (var i = 0; i < selector.length; i++) {
        this[i] = selector[i];
    }
    this.length = selector.length;
    return this;
} else if (typeof selector == 'function') {
    Kodo.ready(selector);
    return;
}
```

首先selector可能为object的情况,比如传入的是原生dom对象,dom数组对象. 另外要记得转为数组`var selector = [selector];

因为有可能是一个元素比如是window,document等否则没法循环

然后selector如果是function那我们就认为他是domReady

PS:在这我判断的并没有非常的全面,仅仅具备了基础功能

```
Kodo.ready = function(fn) {

    doc.addEventListener('DOMContentLoaded',function() {
        fn && fn();
    },false);
    doc.removeEventListener('DOMContentLoaded',fn,true);

};
```

然后这个是ready的源码,由于我们只兼容高端浏览器所以仅仅需要这样写即可.

既然你都看到这了,还不给我一个star说得过去么你!! :(

Lesson-4

===

这个版本我们要增加一个用的非常多的方法!

那就是each!

我们知道each不仅能遍历数组,还能遍历对象.

首先我们需要一个对数组进行验证的方法

```
function isArray(obj) {
    return Array.isArray(obj);
}
```

接着就是我们的重头戏

```
Kodo.each = function(obj,callback) {
    var len = obj.length,
        constru = obj.constructor,
        i = 0;

    if(constru === window.f) {
        for (; i < len; i++) {
            var val = callback.call(obj[i],i,obj[i]);
            if(val === false) break;
        }
    } else if (isArray(obj)) {
        for (; i < len; i++) {
            var val = callback.call(obj[i],i,obj[i]);
            if(val === false) break;
        }
    } else {
        for( i in obj ) {
            var val = callback.call(obj[i],i,obj[i]);
            if(val === false) break;
        }
    }

};
```

因为我们还可能遍历Kodo数组对象

如

```
f("div").each(function(index,item) {

})
```

所以还需要一个判断 是否是Kodo数组对象

```
if(constru === window.f) {
    for (; i < len; i++) {
        var val = callback.call(obj[i],i,obj[i]);
        if(val === false) break;
    }
}
```

在这应该强调下call的用法,还是很多人不知道call何时使用.

在我们的callback里 第一个参数是下标,第二个参数是当前的对象,然后this还要指向他自己

所以 `callback.call(obj[i],i,obj[i]);` 就是这样写 第一个参数是改变this指向,第二个参数是下标,第三个是自己本身

很简单不是吗?

既然你都看到这里了,还不给我一个star?!

Lesson-5

===

这个版本新增6个方法,find(),first(),last(),eq(),get(),ajax

先给出代码

```
find : function(selector) {
    if(!selector) return;
    var context = this.selector;
    return new Kodo(context + ' ' + selector);
},
first : function() {
    return new Kodo(this[0])
},
last : function() {
    var num = this.length - 1;
    return new Kodo(this[num]);
},
eq : function(num) {
    var num = num < 0 ? (this.length - 1) : num;
    console.log(num);
    return new Kodo(this[num]);
},
get : function(num) {
    var num = num < 0 ? (this.length - 1) : num;
    console.log(num);
    return this[num];
}
```

我们要仔细分辨下,这4个方法在jQuery中返回的都是什么对象?到底是dom对象还是jQuery对象.

明白了这个后就很容易能写出这4个方法

```
find : function(selector) {
    if(!selector) return;
    var context = this.selector;
    return new Kodo(context + ' ' + selector);
}
```

首先find, 我们知道一般都会这样写 $('div').find('span') 查找div下的span,返回的是span数组对象,而不是原生的dom对象

那么我们就可以换个思路,因为我们能拿到 $('div') 这个selector对吧? 也就是 div

既然又要find('span'),我们的selector就可以写成 ('div span'),之后直接返回新的数组对象不就好了吗??

`var context = this.selector;` 先缓存当前的selector上下文,之后拼接我们find的selector

所以最后return 就变为 `new Kodo(context + ' ' + selector);` 虽然效率不一定高,总是一种解决思路不是吗?

```
first : function() {
    return new Kodo(this[0])
},
last : function() {
    var num = this.length - 1;
    return new Kodo(this[num]);
},
eq : function(num) {
    var num = num < 0 ? (this.length - 1) : num;
    console.log(num);
    return new Kodo(this[num]);
},
get : function(num) {
    var num = num < 0 ? (this.length - 1) : num;
    console.log(num);
    return this[num];
}
```

find方法比较难解决,之后这4个就很容易了,first,last,eq,分别返回的都是封装后的对象,只有get返回的是原生 dom对象.

我们根据之前的思路,直接取数组对象的index,return下新的即可,是不是很简单? :)

之后是ajax部分

之前说过之所以,可以用 `$.ajax` 直接调用,是因为可以把方法直接挂在在构造函数上,作为静态方法

所以我们只需要写好ajax最后把你想要公开的接口放在Kodo上即可

```
Kodo.get = function(url,sucBack,complete) {
    var options = {
        url : url,
        success : sucBack,
        complete : complete
    };
    ajax(options);
};
Kodo.post = function(url,data,sucback,complete) {
    var options = {
        url : url,
        type : "POST",
```

```
            data : data,
            sucback     : sucback,
            complete : complete
        };
        ajax(options);
    };
    function ajax(options) {
        var defaultOptions = {
            url: false, //ajax 请求地址
            type : "GET",
            data : false,
            success: false, //数据成功返回后的回调方法
            complete: false //ajax完成后的回调方法
        };
        for (var i in defaultOptions) {
            if (options[i] === undefined) {
                options[i] = defaultOptions[i];
            }
        }
        var xhr = new XMLHttpRequest();
        var url = options.url;
        xhr.open(options.type, url);
        xhr.onreadystatechange = onStateChange;
        if (options.type === 'POST') {
            xhr.setRequestHeader('Content-Type', 'application/x-www-for
        }
        xhr.send(options.data ? options.data : null);

        function onStateChange() {
            if (xhr.readyState == 4) {
                var result,
                    status = xhr.status;

                if ((status >= 200 && status < 300) || status == 304) {
                    result = xhr.responseText;
                    if (window.JSON) {
                        result = JSON.parse(result);
                    } else {
                        result = eval('(' + result + ')');
                    }
                    ajaxSuccess(result, xhr)
                } else {
                    console.log("ERR", xhr.status);
                }
            }
        }
        function ajaxSuccess(data, xhr) {
            var status = 'success';
            options.success && options.success(data, options, status, x
            ajaxComplete(status)
        }
        function ajaxComplete(status) {
            options.complete && options.complete(status);
```

```
        }
    }
```

在这我就不细讲ajax的具体过程,我也实现了一个比较简单的ajax,就公开了get和post方法.大家可以实现一个更加复杂好用的ajax替换我这段代码

你说你都耐心的翻到这了? 不给我一个star说的过去么你?

Lesson-6

===

这个版本完善hasClass和css 方法.

新增 attr和data

```
css: function(attr, val) { //链式测试

    for (var i = 0; i < this.length; i++) {
        if(typeof attr == 'string') {
            if (arguments.length == 1) {
                return getComputedStyle(this[0], null)[attr];
            }
            this[i].style[attr] = val;
        } else {
            var _this = this[i];
            f.each(attr,function(attr,val) {
                _this.style.cssText += '' + attr + ':' + val + ';'
            });
        }
    }
    return this;
}
```

在我们上一个版本中,没有对css方法传对象进行解析,在这我们要进行完善.

刚刚好我们现在已经有了each方法!直接用上吧!

在我们的for循环中,要先判断下传入的attr参数是字符串还是对象.

如果是字符串,我们就按照 `css('width','100px')` 这样的方式处理

如果是对象 `css({"width":'100px','height':'200px'})`

```
var _this = this[i];
f.each(attr,function(attr,val) {
    _this.style.cssText += '' + attr + ':' + val + ';';
});
```

首先我们缓存下当前的this,然后用cssText方法,直接拼接进去即可.

接着我们需要完善hasClass方法.这里要着重说明下!目前我搜到的一大堆hasClass方法与jQuery的实现都是不同的

比如有这样的dom结构

```
<div id="pox">
    <ul>
        <li class="a c">pox</li>
        <li class="b">pox</li>
        <li>pox</li><li>pox</li>
        <li>pox</li>
    </ul>
</div>
```

我们如果写$('#pox li').hasClass('b')与$('#pox li').hasClass('a')那都会是什么样的结果呢?

结果是都会返回true.

而现在基本能搜到的完全没有做这方面的判断.所以我们来看看我是如何实现的

```
hasClass : function(cls) {
    var reg = new RegExp('(\\s|^)' + cls + '(\\s|$)');
    var arr = [];
    for (var i = 0; i < this.length; i++) {
        if (this[i].className.match(reg)) {
            return true;
        }
    }
    return false;
}
```

首先我们需要一个正则匹配,还需要一个数组,进行存储每个元素是否有存在判断的class

然后我们再在那个数组中寻找是否有true?如果有true,则返回true,如果一个true都没有的情况下,才能完全返回false.希望大家在这里要注意以下

最后是我们的attr和data方法

```
attr : function(attr, val) {
    for (var i = 0; i < this.length; i++) {
        if(typeof attr == 'string') {
            if (arguments.length == 1) {
                return this[i].getAttribute(attr);
            }
            this[i].setAttribute(attr,val);
        } else {
            var _this = this[i];
            f.each(attr,function(attr,val) {
                _this.setAttribute(attr,val);
            });
        }
    }
    return this;
},
data : function(attr, val) {
    for (var i = 0; i < this.length; i++) {
        if(typeof attr == 'string') {
            if (arguments.length == 1) {
                return this[i].getAttribute('data-' + attr);
            }
            this[i].setAttribute('data-' + attr,val);
        } else {
            var _this = this[i];
            f.each(attr,function(attr,val) {
                _this.setAttribute('data-' + attr,val);
            });
        }
    }
    return this;
}
```

这两个方法就很简单啦,跟CSS方法类似,先判断第一个参数是否为字符串,如果是字符串就是直接增加一个属性.如果是对象,就each下一个一个set即可.

毛主席说过,只阅不star都是耍流氓! :(

Lesson-7

===

新增 html,text,append,before,after,remove

```
html: function (value) {
    if (value === undefined && this[0].nodeType === 1) {
        return this[0].innerHTML;
    } else {
        for (var i = 0; i < this.length; i++) {
            this[i].innerHTML = value;
        }
    }
    return this;
},
text: function (val) {
    if (val === undefined && this[0].nodeType === 1) {
        return this[0].innerText;
    } else {
        for (var i = 0; i < this.length; i++) {
            this[i].innerText = val;
        }
    }
}
```

`html()` 方法我就用了这种很愚蠢的方法实现了!比起之前的data,attr,css什么的简单多了,大家可以自己继续完善.

接着是我们的重头戏,3个文档插入.我找到了一个原生js叼叼的方法

`insertAdjacentHTML` 来让我们去看下MDN是怎么解释的

概述

> insertAdjacentHTML() 将指定的文本解析为 HTML 或 XML，然后将结果节点插入到 DOM 树中的指定位置处。该方法不会重新解析调用该方法的元素，因此不会影响到元素内已存在的元素节点。从而可以避免额外的解析操作，比直接使用 innerHTML 方法要快。

语法

`element.insertAdjacentHTML(position, text);` `position` 是相对于 `element` 元素的位置，并且只能是以下的字符串之一：

`beforebegin` 在 `element` 元素的前面。 `afterbegin` 在 `element` 元素的第一个子节点前面。 `beforeend` 在 `element` 元素的最后一个子节点后面。 `afterend` 在 `element` 元素的后面。 `text` 是字符串，会被解析成 `HTML` 或 `XML` ，并插入到 DOM 树中。

兼容性

Chrome	Firefox	IE	Opera	Safari
1.0	8.0	4.0	7.0	4.0

简直神器有木有?!

所以我们写一个这样的方法吧!

```
function domAppend(elm, type, str) {  //实现append、after、before操作
    elm.insertAdjacentHTML(type, str);
}
```

然后只需要传对应参数就好了!如此简单

```
append: function (str) {
    for (var i = 0; i < this.length; i++) {
        domAppend(this[i], 'beforeend', str);
    }
    return this;
},
before: function (str) {
    for (var i = 0; i < this.length; i++) {
        domAppend(this[i], 'beforeBegin', str);
    }
    return this;
},
after: function (str) {
    for (var i = 0; i < this.length; i++) {
        domAppend(this[i], 'afterEnd', str);
    }
    return this;
}
```

接着是remove方法,在这我只做删除自身节点,就没继续拓展了.大家可以自行完善

```
remove: function () { //只能删除自身
    for (var i = 0; i < this.length; i++) {
        this[i].parentNode.removeChild(this[i]);
    }
    return this;
}
```

您给予的star,就是给代码赋予灵魂.

Lesson-8

===

事件机制

在讲事件机制之前呢,我们有一个很重要的东西要先讲,那就是如何实现事件委托(代理).

只有必须先明白了如何实现一个事件委托,我们才能更好的去实现on和off.在我看来,on和off里最难实现的就是他的事件委托.

```
function delegate(agent,type,selector,fn) {
    agent.addEventListener(type,function(e) {

        var target = e.target;
        var ctarget = e.currentTarget;
        var bubble = true;

        while(bubble && target != ctarget) {
            if(filiter(agent,selector,target)) {
                bubble = fn.call(target,e);
                return bubble;
            }
            target = target.parentNode;
        }
    },false);
    function filiter(agent,selector,target) {
        var nodes = agent.querySelectorAll(selector);
        for (var i = 0; i < nodes.length; i++) {
            if (nodes[i] == target) {
                return true;
            }
        }
    }
}
```

以上是我对整个委托的实现,当然在这只做了非常简单的实现,没有对很多别的情况进行判断,也没有多个参数是否捕捉.

我们先拆解下分析.

```
function filiter(agent,selector,target) {
    var nodes = agent.querySelectorAll(selector);
    for (var i = 0; i < nodes.length; i++) {
        if (nodes[i] == target) {
            return true;
        }
    }
}
```

先看这个方法,这其实就是一个元素过滤,作用就是为了过滤出我们委托的元素具体是谁.target就是我们具体的委托元素

```
agent.addEventListener(type,function(e) {

    var target = e.target;
    var ctarget = e.currentTarget;
    var bubble = true; //是否阻止冒泡

    while(bubble && target != ctarget) {
        if(filiter(agent,selector,target)) {
            bubble = fn.call(target,e);
        }
        target = target.parentNode;
        return bubble;
    }
},false);
```

然后是我们的主要部分.其实这里就很简单,while的条件判断两个,第一个是是否阻止冒泡,第二个判断是冒泡是否到顶.

接着我们进行filiter进行过滤,如果返回true,则是我们的委托元素,直接call即可.

如果不做过多的兼容处理,实现一个委托还是比较容易的.

PS:如果您还是不太明白,可以来这看更具体的解释.http://www.meckodo.com/?p=309

您的star是检验代码的唯一标准!:)

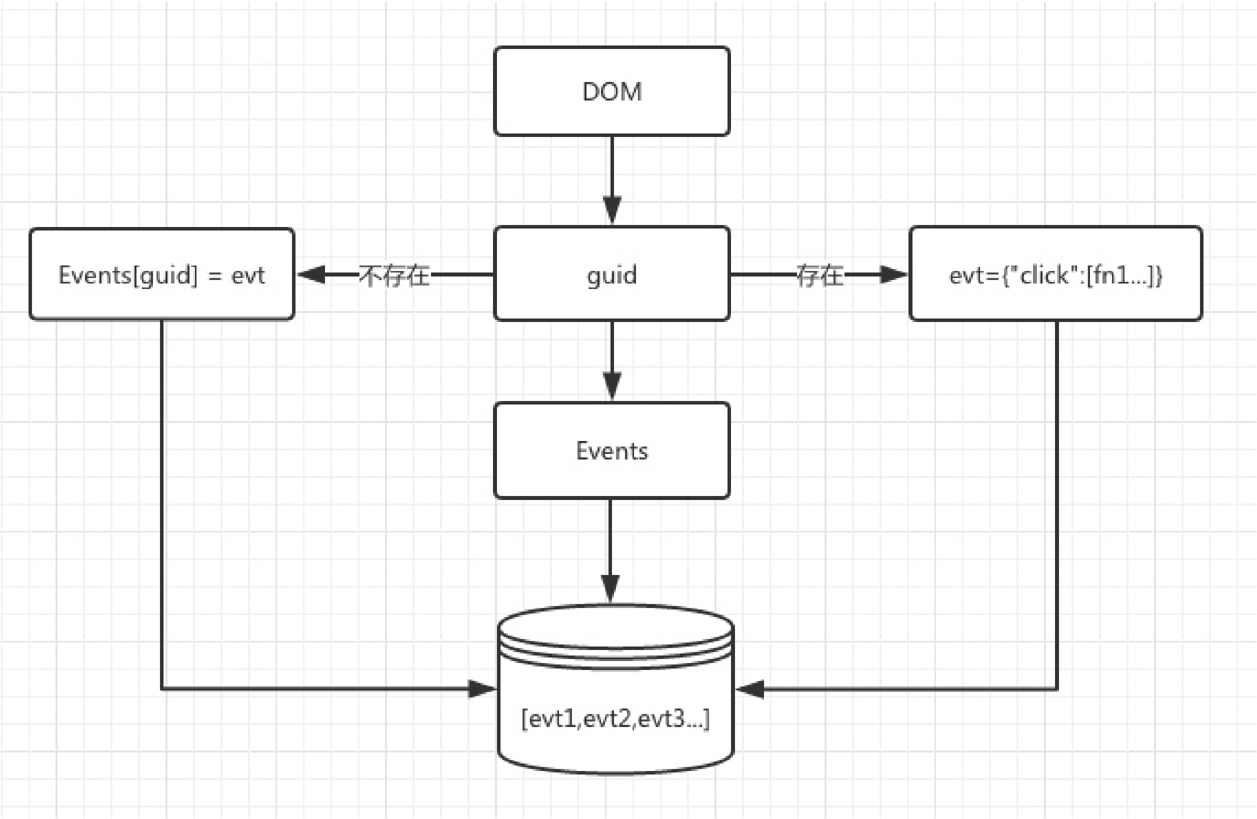
Lesson-9

关于事件部分,我思考了很久,也参考了许多,到底如何能用一个很简单的方法实现一模一样的on、off呢?

最后我的设计思路是:

> 1.有一个全局存储所有 `Events` 的数组，存放每个 `dom` 元素上的事件。
>
> 2.给每个 `DOM` 一个 `guid` 的唯一标识符，通过这个 `guid` 来找出 `Events` 数组里的事件。

由于逻辑比较复杂，我们先来画个图看看。



首先，我们利用DOM可以增加自定义属性的原理，在它的身上存一个guid。

之后整个事件机制就根据这个guid来进行查找与存储。

接下来是代码部分

```
Kodo.Events = []; //事件绑定存放的事件
Kodo.guid = 0; //事件绑定的唯一标识

on: function(type, selector, fn) {
    if (typeof selector == 'function') {
        fn = selector; //两个参数的情况
        for (var i = 0; i < this.length; i++) {
            if (!this[i].guid) {
                this[i].guid = ++Kodo.guid;
                //guid 不存在，给当前dom一个guid

                Kodo.Events[Kodo.guid] = {};
                /*
                *给Events[guid] 开辟一个新对象
                *用于存储这个dom上的所有事件方法
                */

                Kodo.Events[Kodo.guid][type] = [fn]; //每个方法都是一
                //给这个新对象，赋予事件数组 "click" : [fn1,fn2,...]

                bind(this[i], type, this[i].guid);//绑定事件

            } else {//guid存在的情况
                var id = this[i].guid;
                if (Kodo.Events[id][type]) {
                    //如果这存在是当前事件已经存过，不用在绑定事件,直接放入
                    Kodo.Events[id][type].push(fn);
                } else {
                    //这是存新事件，所以需要重新绑定一次
                    Kodo.Events[id][type] = [fn];
                    bind(this[i], type, id);
                }
            }
        }
    }
}

function bind(dom, type, guid) {
    dom.addEventListener(type, function(e) { //绑定相应事件
        for (var i = 0; i < Kodo.Events[guid][type].length; i++) {
            //循环执行那个方法数组即可
            Kodo.Events[guid][type][i].call(dom, e); //正确的dom回调
        }
    }, false);
}
```

由于方法过长，我就把讲解的都写在了代码里，这样看的也会更方便一些。

代码还是不够形象！我们来看看log就能更清晰明白其中的奥秘。

通过控制台log出 `f.Events` 发现正是我们想要的结果，每个 `dom` 对应一个自己的 `evtObj` ，通过 `Kodo.Events[guid]` 可以得到指定的 `evtObj` 。然后即可取出自己相应的事件。

```
<div id="box" style="color: red;">
    <ul>
        <li class="testLi">11</li>
        <li class="testLi">12</li>
        <li class="testLi">13</li>
        <li class="testLi">14</li>
    </ul>
</div>
f('#box li').on('click',function(e) {
    console.log(this.innerHTML);
});
```

```
> f.Events
< [undefined × 1, ▼Object          , ▼Object          , ▼Object          , ▼Object          ]
                   ▶click: Array[1]   ▶click: Array[1]   ▶click: Array[1]   ▶click: Array[1]
                   ▶__proto__: Object ▶__proto__: Object ▶__proto__: Object ▶__proto__: Object
> |
```

如果我继续新增事件

```
f('#box li').eq(0).on('click',function(e) {
    console.log('我只有第一个li才会触发');
});
f('#box li').eq(0).on('touchstart',function(e) {
    console.log('我只有第一个li才会触发');
});
```

```
> f.Events
< [undefined × 1, ▼Object              , ▼Object          , ▼Object          , ▼Object          ]
                   ▶click: Array[2]       ▶click: Array[1]   ▶click: Array[1]   ▶click: Array[1]
                   ▶touchstart: Array[1]  ▶__proto__: Object ▶__proto__: Object ▶__proto__: Object
                   ▶__proto__: Object
>
```

可以发现，我只针对于第一个li增加了事件。log出Evnets也就只有第一个Object有新增，并且会增加到对应的事件数组里。

理解了这个后要解除事件绑定，那就非常简单了。同样根据guid查找到对应的方法数组，delete即可

```
off: function(type, selector) {
    if (arguments.length == 0) {
        //如果没传参数，清空所有事件
        for (var i = 0; i < this.length; i++) {
            var id = this[i].guid;
            for (var j in Kodo.Events[id]) {
                delete Kodo.Events[id][j];
            }
        }
    } else if (arguments.length == 1) {
        //指定一个参数，则清空对应type的事件
        for (var i = 0; i < this.length; i++) {
            var id = this[i].guid;
            delete Kodo.Events[id][type];
        }
    }
}
```

一个没有带有事件委托的on、off就可以这样实现了。

那如果我们要实现带委托的怎么办呢？

我们可以用这同样的思路实现，只是要多进行一个指定selector的存储。

这个我们就放在下一课最后讲解。

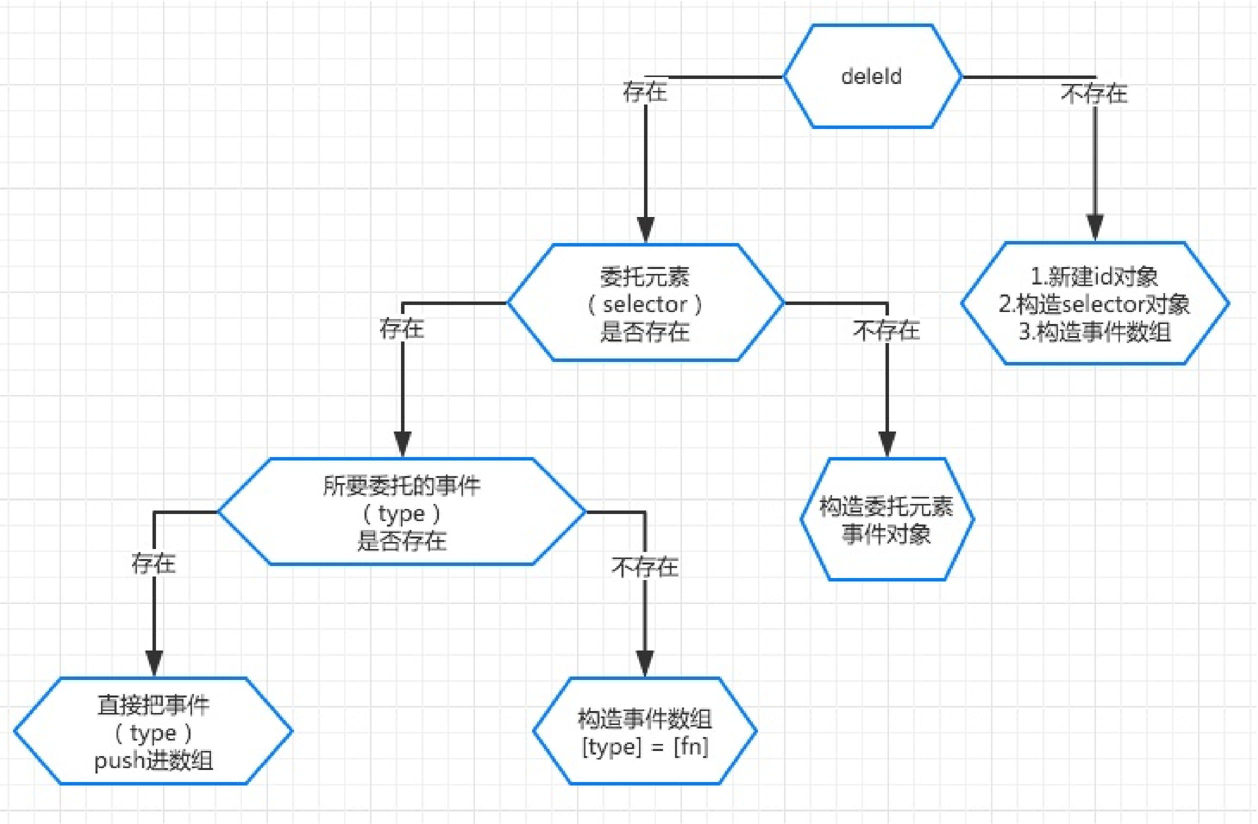
> star是尊重作者知识果实最好的回报 :)

Lesson-10

实现on,off的事件委托!

我们能根据之前的思路,利用同样的方法实现一个事件委托.

先来看看流程图



然后先看看结果是如何，毕竟流程图看的也不一定能懂。

最后我们再来看看代码

```
Kodo.deleEvents = []; //事件委托存放的事件
Kodo.deleId = 0; //事件委托的唯一标识

on: function(type, selector, fn) {
    if (typeof selector == 'function') {
        fn = selector; //两个参数的情况
        //事件绑定过程
    } else {
        //事件委托过程
        for (var i = 0; i < this.length; i++) {
            if ( !this[i].deleId ) {
                this[i].deleId = ++Kodo.deleId;
                //同样是判断是否有唯一id

                Kodo.deleEvents[Kodo.deleId] = {};
                //没有则创建id对象 也就是f.deleEvents[]新开辟一个新对象

                Kodo.deleEvents[Kodo.deleId][selector] = {};

                //构造 selector对象
                /*
                *   如 Kodo.deleEvents[1] =
                *   {
```

```
                 *       "#box li" : {},
                 *        "#pox" : {}
                 *    }
                 */

                Kodo.deleEvents[Kodo.deleId][selector][type] = [fn]
                //构造我们的事件数组
                /*
                 *   如 Kodo.deleEvents["#box li"] =
                 *   {
                 *       "click" : [fn1,fn2...],
                 *        "touchstart" : [fn1,fn2....]
                 *    }
                 */
                delegate(this[i],type,selector);
                //用委托的方式进行绑定
            } else {
                //如果id存在的情况
                var id = this[i].deleId,
                    position = Kodo.deleEvents[id];//委托元素的事件存

                if(!position[selector]) {
                    //先判断如果selector存储的对象不存在
                    position[selector] = {};
                    //新建selector对象 (与上面的selector构造相同)
                    position[selector][type] = [fn];
                    //构造事件数组对象   (与上面的type构造相同)

                    delegate(this[i],type,selector);
                    //因为是新的selector 所以要再绑定
                } else {
                    //selector 存储对象存在的情况
                    if ( position[selector][type] ) {
                        //如果事件数组已经有了，则直接push进来
                        position[selector][type].push(fn);

                    } else {
                        //如果事件数组没有，那就构造事件数组
                        position[selector][type] = [fn];

                        //因为是新的绑定的事件，所以要重新绑定
                        delegate(this[i],type,selector);
                    }

                }
            }
        }
    }
},
```

继续再看一遍log的结果，对比刚刚的代码

连同代码，我在注释里已经非常的详细解释了整个过程，大家结合控制台log的结果，在看看最初的流程图结合的看，我相信有点点耐心就能马上理解了。

绑定过程都会比较复杂，理解了绑定过程后，下面off的实现就很容易了。

off本身是可以传2个参数的，第一个参数为事件type，第二个参数是委托元素selector

```
off: function(type, selector) {
    if (arguments.length == 0) {
        //如果没传参数，清空所有事件

    } else if (arguments.length == 1) {
        //指定一个参数，则清空对应的事件

    } else {
        //直接根据dom上存有的deleId，找到对应的deleEvents里的位置
        //删除委托元素上的type事件数组即可
        for (var i = 0; i < this.length; i++) {
            var id = this[i].deleId;
            delete Kodo.deleEvents[id][selector][type];
        }
    }
}
```

最后看看我们修改过后的 delegate方法

```
function delegate(agent, type, selector) {
    var id = agent.deleId; //先获取被委托元素的deleId
    agent.addEventListener(type, function(e) {
        var target = e.target;
        var ctarget = e.currentTarget;
        var bubble = true;

        while (bubble && target != ctarget) {
            if (filiter(agent, selector, target)) {
                for (var i = 0; i < Kodo.deleEvents[id][selector][t
                    bubble = Kodo.deleEvents[id][selector][type][i]
                    //循环事件数组 直接call
                }
            }
            target = target.parentNode;
            return bubble;
        }
    }, false);

    function filiter(agent, selector, target) {
        //过滤函数
    }
}
```

这里修改的就只有二个地方

1.获取被委托元素的deleId，因为我们整个委托机制都与他有关。 2.通过id在deleEvents里查找对应的事件数组，循环执行即可

以上就是整个委托的过程！

```
f("you").on('star','me',function(){
    console.log('success!');
});
```

Lesson-11

新增width,height,extend

事件部分讲完了后,我们最后实现3个方法.

```
width : function(w) {
    if(arguments.length == 1) {
        for (var i = 0; i < this.length; i++) {
            this[i].style.width = w + 'px';
        }
    } else {
        if (this[0].document === doc ) {
            return this[0].innerWidth;
        } else if (this[0].nodeType === 9 ){
            return document.documentElement.clientWidth;
        } else {
            return parseInt(getComputedStyle(this[0],null)['width']
        }
    }
},
```

关于width(),height()就常用的就2种,一个是取值,一个是赋值.

我们通过判断arguments的个数,是取值还是赋值.

赋值很容易,我们就用最简单的办法,直接设置.

如果是取值,那我们就要做个判断,因为window,和document的取法是不一样的.

还有一种可能性是,当dom元素的display为none的时候,直接取是取不到的.在这我就不做处理了.大家思考一下可以自己尝试.

思路是把dom设置为position:absolute;visible:hidden;然后取,在设置回去.

同理height方法也是如此.我就直接给出代码了

```
height : function(h) {
    if(arguments.length == 1) {
        for (var i = 0; i < this.length; i++) {
            this[i].style.height = h + 'px';
        }
    } else {
        if(this[0].document === doc ) {
            return this[0].innerHeight;
        } else if (this[0].nodeType === 9 ){
            return document.documentElement.clientHeight;
        } else {
            return parseInt(getComputedStyle(this[0],null)['height
        }
    }
}
```

两者几乎相同只是改了API,其实完全可以封装为一个方法复用.

---

jQuery之所以那么广受大众所爱,还一个非常重要的就是他的extend方法.如果没有了他,将不会有现在那么多jQuery插件的诞生

在此,我们就实现一个非常简单的浅拷贝.(然而jQuery的extend非常强大)

```
Kodo.prototype.extend = Kodo.extend = function() {
    var options = arguments[0];
    for( var i in options) {
        this[i] = options[i];
    }
};
```

这个方法我们不仅要能拓展静态方法,也要能拓展实例方法.

所以 `Kodo.prototype.extend = Kodo.extend =` 直接这样写了.

里面内容过于简陋就不过多讲解了 :)

然后我们就能这样拓展我们的插件了

```
f.prototype.extend({  //实例方法
    alert : function(msg) {
        alert(msg)
    }
});
f.extend({ //静态方法
    alert : function(msg) {
        alert(msg)
    }
});

f.alert('123');//调用
f("div").alert('123');//调用
```

其实jQuery还有很多别的部分,比如队列,动画,异步.都是一些非常值得自己去实践尝试的.

但至此,我们的小轮子基本也就完结了

另外的手势番外篇,本想直接集成在这里面.如果有大众所需,我就继续更下去

您连11节的课程都有耐心看完,何必不顺手点下右上角的star呢? >.<